Distributed and shared memory parallelism with a smoothed particle hydrodynamics code

Richard J. Goozée* Peter A. Jacobs†

(Received October 1, 2001)

Abstract

The Smoothed Particle Hydrodynamics (SPH) method, being Lagrangian in nature, provides advantages in modelling flows containing interfaces. The SPH method is completely mesh-free, modelling the fluid as a collection of N particles which move with the fluid velocity. The continuum fluid properties at a particular location are interpolated as weighted sums of the properties of surrounding particles in a process known as kernel interpolation. In its simplest form the SPH method requires that every particle is used in the updating of every other particle. This leads to a solution

*Centre for Hypersonics, The University of Queensland, AUSTRALIA. mailto:goozee@mech.uq.edu.au

†Centre for Hypersonics, The University of Queensland, AUSTRALIA. mailto:peterj@mech.uq.edu.au

[0]See http://anziamj.austms.org.au/V44/CTAC2001/Gooz for this article,
 Published 1 April 2003. ISSN 1446-8735

time that is proportional to N^2. Methods using only nearby particles reduce this requirement significantly; however, the SPH method is still expensive and would benefit from being implemented on a parallel computer. An example SPH code has been parallelised using MPI, OpenMP and BSP, and its performance has been measured on an SPH, Origin 2000 and a Beowulf workstation cluster.

Contents

1 Introduction

High speed flows are of importance in aircraft flight and for access to and from space. The Centre for Hypersonics at the University of Queensland conducts experiments in the hypersonic flow regime using impulse facilities such as the T4 free-piston shock tunnel and the X3 superorbital expansion tube. One of our aims is to build computational models of the flows in these facilities in order to support the experimental efforts. These supersonic and hypersonic flows contain strong shocks and expansions and have appreciable gradients of density through the fluid.

The flows in shock tunnels and expansion tubes involve a number of moving interfaces between different gases including the contact surface and the flow around the rupture of the diaphragm. These interfaces are important as they have a significant effect on the quality of the test flow and, therefore, on the quality of the experimental data. The time that tests can be run for in these facilities is typically very short and depends on how long a homogeneous flow of test gas can be maintained. A flow simulation method that can accurately represent these interfaces without introducing significant amounts of diffusion in doing so would be of benefit.

The Computational Fluid Dynamics (CFD) methods that are available can be broadly classified as being based on two descriptions of fluid motion: the Eulerian description and the Lagrangian description. Meshes are typically used to join computational points for the purpose of interpolating fluid properties. With the Eulerian description it is the geometry of the flow domain that is discretised and the fluid flows through the stationary discrete elements. With the Lagrangian description it is the fluid itself that is divided into discrete elements which then move with the fluid.

The majority of computational methods that have been used for detailed modelling of impulse facility flows are Finite Volume methods [6] based on the Eulerian description; however, the Lagrangian description could provide advantages in cases where modelling fluid interfaces is important. Methods that are Lagrangian in nature do not require high order advection schemes or explicit interface reconstruction, because the interface is maintained through the movement of the elements of fluid. These methods, however, suffer problems caused by distortion of the mesh. It is possible to continually remap the Lagrangian mesh, but this would introduce numerical diffusion and, therefore, would reduce the advantage of being able to advect interfaces with little diffusion.

The problem of mesh distortion can be avoided by not using meshes at all. What is required for this is a method of interpolating fluid properties without relying on a mesh joining computational points. One such method that uses this approach is the Smoothed Particle Hydrodynamics (SPH) method [3, 7]. SPH is a meshless, purely Lagrangian numerical method for the transient solution of the Euler equations in which continuum fluid is represented by a collection of fluid particles. The Lagrangian form of the conservation equations of fluid flow become the equations of motion of these interpolation points as they move with the flow. In modelling a fluid interface with SPH the sharp interface is smeared out by the interpolating function. This will reduce detail around the interface; however, since these smoothed computational elements move with the interface, an improved representation may still be obtained.

The approach we take to the modelling of interfaces in complicated flows requires the use of large three dimensional simulations; this means that the performance of the method is important. The basic solution procedure using SPH is, in essence, proportional to N^2, which results from the case where each of the N particles

in the simulation must calculate its properties using information from every other particle in the simulation. This degree of scaling means that large simulations would be prohibitively expensive and three dimensional simulations with sufficient resolution would be infeasible. As an example, a two dimensional simulation in which the initial arrangement of 100×100 particles, taking 15 minutes to solve, would take approximately 100 days to solve for a three dimensional arrangement of $100 \times 100 \times 100$ particles. There are techniques specific to SPH to reduce this requirement without invalidating the solution. One of the most important of these involves dividing the solution domain into regions that can then be used to identify neighbouring particles to be included in the computation; only those particles nearby have a significant effect on a particle behaviour.

The other important method used for implementing larger SPH simulations is parallelisation. In a parallel SPH simulation the work of interpolating the flow field and updating the properties of particles is divided amongst several processing elements. In the present study, this was done with an example SPH code using OpenMP, MPI and BSP. The parallel code, developed with the three methods, was used in solutions of size ranging from 6,250 to 200,000 particles, and each was run with one, two, four and eight processors. The performance was measured and compared to the sequential performance of the code. These results are used to investigate general trends in the code's performance, both as the size of the simulations is scaled, and as the number of processors is scaled.

2 A simple SPH code

An SPH simulation is started by placing the particles in the domain such that the density of the fluid is represented by the relative density of the particle positions. To interpolate the information that we have at particle positions a technique known as kernel interpolation is used [13]. To do this the information stored at a point in space is numerically distributed over the surrounding area by a kernel function, $W(r)$. The degree of smoothing is defined by a parameter known as the smoothing length and can be either constant for all particles or vary depending on the local density.

The state vector of particle properties is $\{\mathbf{r}, \mathbf{v}, e\}$, where $\mathbf{r}$ is the position vector, $\mathbf{v}$ is the velocity vector, and e is the specific internal energy. The time derivatives of the state vector are $\{\mathbf{v}, d\mathbf{v}/dt, de/dt\}$.

Equation (1) shows the summation used for some fluid property, A, being interpolated at the sample point $\mathbf{r}$ using the kernel smoothing function $W(\mathbf{r}-\mathbf{r}_b, h)$. The contributions to the properties from all particles b in the domain are summed to produce

$$A_s(\mathbf{r}) = \sum_b A_b \frac{m_b}{\rho_b} W(\mathbf{r} - \mathbf{r}_b, h) \tag{1}$$

where m_b is the mass of fluid assigned to the particle b and ρ_b is the density of the particle b. An important aspect of kernel interpolation is that, with the right choice of kernel, the gradient of fluid properties is interpolated directly using the gradient of the kernel. With the Gaussian kernel this gradient is known analytically and, therefore, can be used to interpolate these gradients in the same way that fluid properties are calculated. In the case of the Euler equations the gradient of pressure is required:

$$\frac{D\mathbf{v}}{Dt} = \frac{-1}{\rho} \nabla p \tag{2}$$

where $\mathbf{v}$ is the velocity vector of the fluid, ρ is the density of the fluid and p is the pressure of the fluid. This equation is solved using the gradient of the kernel interpolant, resulting in Equation (3). This equation determines the acceleration experienced by the particle using the pressure and density properties of the surrounding particles, denoted by b. The symmetric nature of this equation ensures that particle interactions are equal and opposite [8].

$$\frac{d\mathbf{v}_a}{dt} = -\sum_b m_b \left(\frac{p_a}{\rho_a^2} + \frac{p_b}{\rho_b^2} \right) \nabla_a W(\mathbf{r}_a - \mathbf{r}_b, h) \,. \tag{3}$$

The rate of change of specific internal energy is calculated using:

$$\frac{de_a}{dt} = \frac{1}{2} \sum_b m_b \left(\frac{p_a}{\rho_a^2} + \frac{p_b}{\rho_b^2} \right) \mathbf{v}_a \cdot \nabla_a W(\mathbf{r}_a - \mathbf{r}_b, h) \,. \tag{4}$$

Dissipation in the form of two types of artificial viscous pressures are usually added to Equations (3) and (4): a Von Neumann-Richtmyer artificial viscosity and a bulk viscosity. Pseudo code for the SPH technique is shown in Figure 1.

SPH can be computationally expensive; however, there are techniques for increasing its efficiency and, therefore, the size of simulations that can be run in practice. These include the use of efficiently structured code and compiler optimisations, particle sorting methods, such as cells and hierarchical trees, and overall parallelisation of the algorithm.

On modern computers, effective use of data pipelining, memory hierarchy and, with some processors, superscalar operation give significant improvements in performance. Most systems provide a "fast" compiler tag (such as `-Ofast=ip27` on the SGI Origin 2000 systems) that specify all optimisations to the compiler, which can result in performance improvements as large as 20%. These aspects

1. assign particles to flow domain according to initial density
2. assign initial properties to particles
3. repeat until solution time has been reached
   (a) calculate densities at particle positions
   (b) calculate particle pressures and local sound speeds
   (c) calculate particle accelerations
   (d) calculate rate of change of internal energy
   (e) integrate particle properties forward through time step

FIGURE 1: Pseudo code for the SPH method

of performance are common to all areas of high performance computing.

There are optimisations specific to the SPH method that significantly impact performance. Polynomials are often used for the interpolating kernels instead of Gaussian functions as they are less costly to evaluate. This significantly improves performance as the kernel function must be calculated for every particle's contribution to the summations used. Forces between particles are equal and opposite and so the number of interactions actually calculated in the solution are reduced by a factor of two. This optimisation is easily realised with a sequential solution, but care must be taken when implementing this in parallel.

Particles that are a long distance from the particle being updated provide a negligible contribution to the summation. Special interpolating kernels recognise this and provide compact support [2]:

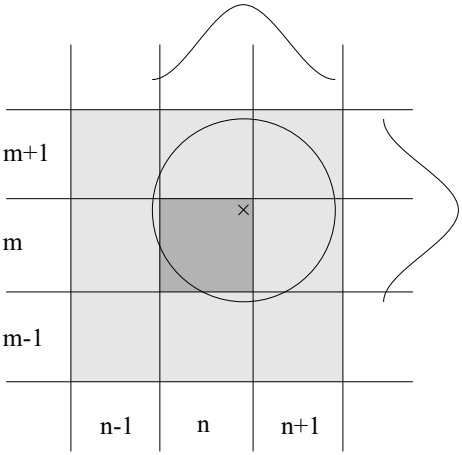

FIGURE 2: Cell sorting used for finding neighbouring particles to those in cell (m, n) with constant smoothing length

outside of a certain radius the contribution is zero. Thus only nearby particles, usually within a radius of two times the smoothing length, need to be considered. Particles are then sorted in order to quickly find their neighbours. For simulations in which the smoothing length does not vary, particles are assigned to fixed-size cells. For a specified location, the nearest cells and, therefore, nearest particles are easily found. Figure 2 shows a particle, marked by a cross, and the circular region in which other particles will have a significant contribution. The cell (m, n) in which the particle lies is shaded and the surrounding cells that are included in the calculation by the sorting technique are lightly shaded.

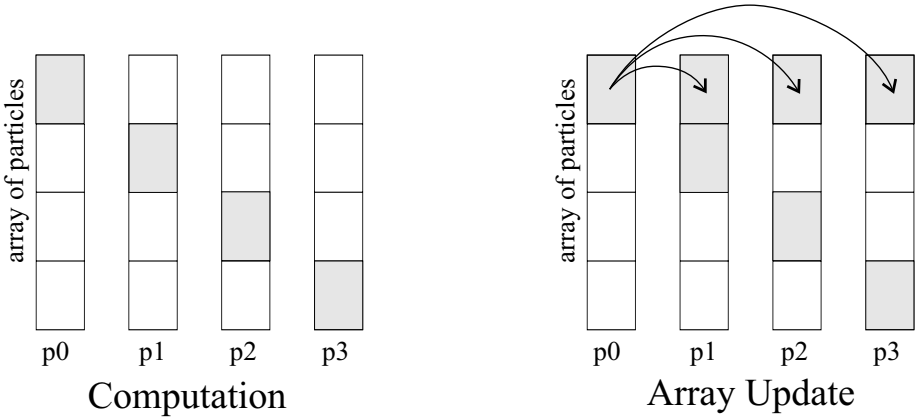

FIGURE 3: Simple parallelisation of an SPH code with four processes

3 Parallel SPH

One of the most important methods for running very large simulations is the use of multiple processors to simultaneously update particle information. For this to be feasible the majority of the computational work needs to be processed independently. With most algorithms, however, sequential regions will remain and some extra work will also be incurred by running an algorithm in parallel. The majority of the work in the SPH method can be parallelised since all particle interactions are independent. Almost all of the computational effort used in the SPH method is in calculating these interactions. The sequential work that remains includes stacking new particles in the array when they are created at inflow boundaries.

Figure 3 shows the simple parallel model for the SPH method. A simulation using four processes is shown. Each process stores its own copy of the array of particles; this copy may, or may not,

be accessible by the other processors. During a time step, each process updates the properties of the particles assigned to it, shown as shaded. At the end of a time step, each process has updated information for its section of the array and out of date information for the other sections. Reconciling these differences will require some data transfer. This data transfer is shown in Figure 3 for the particles associated with process zero. It is a feature of this model that processes can be reading from any particle in the array meaning that the entire array must be updated for each process.

3.1 Choice of parallel architecture

The term supercomputer is used to refer to the most powerful computers available at the time. At present this refers exclusively to parallel computers, as shown in Figure 4. In 1995, single processor and Single Instruction, Multiple Data (SIMD), or vector, computers still featured in the top 500 list; Symmetric Multi-Processor (SMP), or shared memory, computers were firmly established and the new class of Massively Parallel Processor (MPP), or distributed memory, computers were emerging. In 2001 the choice of supercomputer would be broadly stated as being a choice between a shared memory computer and a distributed memory computer. We limit our discussion to these two classifications.

In distributed memory computers, all processors store data in their own memory space. This information is not directly accessible to other processes and must be shared by passing messages containing required information. In contrast, shared memory computers use one memory space that is accessible to all processors. This single memory space simplifies parallelisation by removing the constraint of having to explicitly update data between processes through communication. The shared memory model is limited in

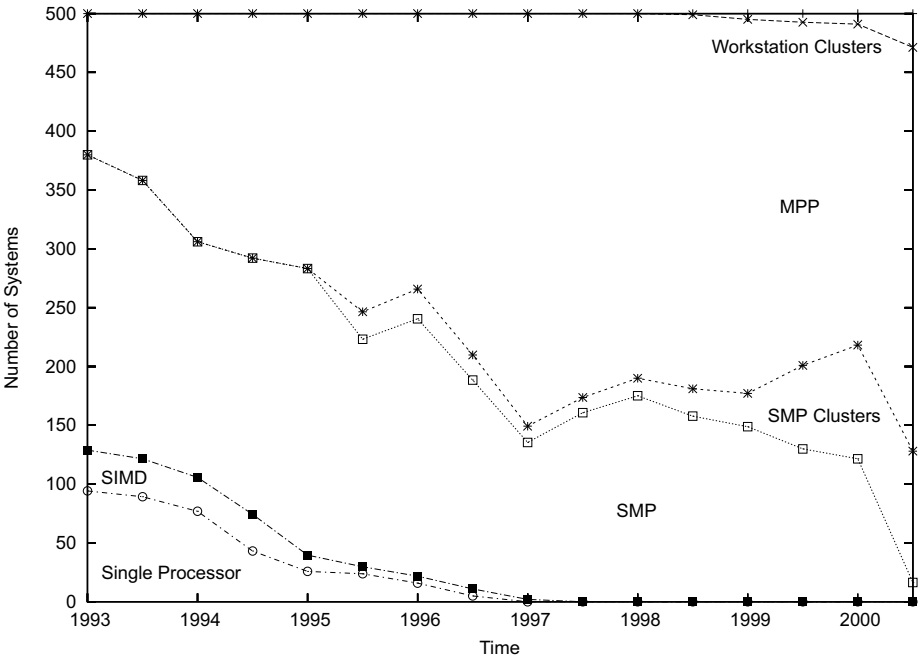

FIGURE 4: The list of the Top 500 supercomputers in the world for the last ten years [10]

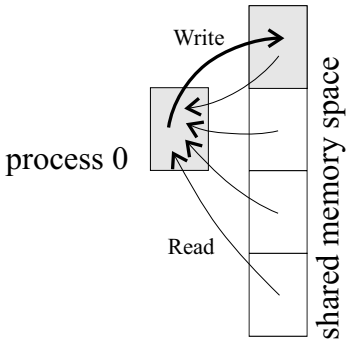

FIGURE 5: Shared Memory Parallelism for process 0

scalability by the need for all processors to practically be able to access the one memory.

Shared memory parallelism with the SPH method only differs from the simple domain parallel method described in Figure 3 in the way that data is stored and shared. Since all processors access the same global memory space there is only one copy of the particle array; the simple model had a copy of the array for each processor. The basic shared memory model is shown in Figure 5 where process 0 is reading the properties of particles from anywhere in the data structure and writing updated results to the section of the global array assigned to it to be updated. The rest of the array is updated by the other processes.

In the distributed memory model each process stores data in its own memory space making it more like the simple parallel model described in Figure 3 than the shared memory model. If processors require updated data that is being stored by another process then this information must be transferred to it by message passing. With message passing, discrete blocks of information are requested and then sent and received explicitly by the two participating processors.

All processors must receive all updated particle information before the start of a new time step since, as with the simple model, updating a particle may require information from any other particle. This results in a large amount of data storage and a large amount of communication that must be performed keeping this data up to date. This method is, however, relatively simple to implement and analyse.

A more advanced method of implementing distributed memory parallelism involves dividing the particles into blocks according to their geometric position rather than their position in the array; this is known as multi-block parallelisation. This has the potential to greatly reduce the amount of communication required as only block boundaries need to be transferred. There are significant complications with this method, however, such as the management of particles rapidly moving between the stationary geometric blocks. This method is not implemented in this paper, but will be investigated further for large simulation sizes.

4 Implementation of P-SPH

The Parallel-SPH code (P-SPH) was implemented using three methods: OpenMP uses the shared memory model and was used on the SGI Origin 2000; the Message Passing Interface (MPI) uses the distributed memory model with message passing and was used on both the Origin 2000 and the Beowulf cluster; and the Bulk Synchronous Parallel (BSP) model which, like MPI, is based on message passing but follows a simpler structure to the parallelism based on supersteps.

4.1 Shared memory (OpenMP)

The OpenMP version of the P-SPH is based on the standard shared memory model of parallelism. The updated data is transferred using memory to memory copying via array access. OpenMP is a specification for a set of compiler directives, library routines and environment variables that can be used to specify shared memory parallelism in Fortran and C programs. Standardisation efforts in shared memory parallel programming are focusing on the development of OpenMP which allows the parallelisation of general regions of code, the nesting of parallel regions inside one another and much of the control over the details of the parallel execution.

As the SGI-OpenMP library used on the Origin 2000 is tuned to this system good performance would be expected. On the computer, underlying hardware and system software takes care of any data transfer between threads giving the impression, to the programmer, that they are using a single memory space. At present OpenMP does not run on distributed memory computers. Such an implementation would perhaps have to be built on top of MPI and would, therefore, incur significant performance penalties.

The performance of OpenMP will be used as the benchmark for parallel performance of P-SPH as it is aimed at being easy to implement and still perform well on the Origin 2000. A major advantage of MPI and BSP over OpenMP is that they run with distributed memory as well as shared memory computers.

The structure of the OpenMP version of P-SPH is the same as that for the sequential version. The difference lies in that when a particular task is performed, such as the calculation of density using kernel interpolation, multiple threads are spawned which all update information for particles assigned to them in the shared memory space. The pseudo-code for the OpenMP version is the same as that

for the sequential version, shown in Figure 1. For steps 4 to 7 the compiler arranges multiple threads to be used in the calculations.

4.2 Message passing interface (MPI)

The MPI version of the code used in the performance investigation in Section 5 is based on the simple model of distributed memory parallelism.

The Message Passing Interface (MPI) has become the standard specification for communication protocols in Multiple Instruction, Multiple Data (MIMD), distributed memory parallel computers [1] with almost all modern supercomputers having MPI libraries available. Also, a lot of the development of MPI has been with workstation clusters in mind and with current trends in supercomputing MPI based applications should be easy to port to new supercomputer facilities.

The basis of message passing is that all CPUs possess a local memory and are able to communicate with the other processes by sending and receiving messages. It is a defining feature of message passing that the sending and receiving of messages are separate operations that must be performed by each process [5]. The complexity of MPI can range from simple send and receive commands to collective communication procedures involving structures of data. The MPI libraries used in this paper complied with version 1.2 of the standard. Three of the most widely used libraries, LAM-MPI [11], MPICH [4] and MPI/Pro [9] were used in the tests on the Beowulf cluster, and the SGI Message Passing Toolkit (MPT) was used on the Origin 2000. The MPT is a library based on the standard that is specially tuned for the Origin 2000 architecture.

The pseudo code for the MPI version of P-SPH is shown in Fig-

1. assign particles to flow domain according to initial density
2. assign initial properties to particles
3. repeat until solution time has been reached
   (a) calculate densities at particle positions
   (b) calculate particle pressures and local sound speeds
   (c) blocking group communication to update particle array
   (d) calculate particle accelerations
   (e) calculate rate of change of internal energy
   (f) integrate particle properties forward through time step
   (g) blocking group communication to update particle array

FIGURE 6: Pseudo code for the MPI version of P-SPH

ure 6. The structure of the pseudo code is not changed from the sequential version, however, the blocking group communications at steps 3c and 3g are added. The densities, pressures and sound speeds calculated at 3a and 3b are required by the calculations at 3d and 3e and so communication is required. When the particles are integrated at 3f this information must be communicated to the other processes before the next time step can commence. With the simple parallel model used, particle information may be required from any position in the particle array and so each process must update the information in every other processors array. The broadcast performed by each process is a blocking communication routine which means that no processes are allowed to proceed past this routine until it is complete. This eliminates the need for an explicit barrier synchronisation at these points.

4.3 Bulk synchronous parallel (BSP)

The Bulk Synchronous Parallel (BSP) version of the P-SPH code is based on the simple model of distributed memory parallelism, much the same as the MPI version. The BSP model of parallel computing was first proposed in 1990 by Valiant [12]. Since then the model has been implemented by a research group at Oxford University producing BSPlib, which is a library of BSP communication and synchronisation primitives. These primitives are callable from both Fortran and C, and the library includes performance analysis, benchmarking and debugging tools. The aim of BSP is to produce a parallel library that is architecture independent, provides scalable parallel performance and yet is conceptually simple. These objectives have largely been achieved in BSPlib, however, results were only obtained for the Origin 2000 since the installation of the library on the Beowulf cluster failed.

Within a superstep, each process performs its independent computations, followed by a global computation phase and then a barrier synchronisation. In this way BSP imposes a simple underlying structure on the parallel code. The superstep-based structure of BSP corresponds well with the time step structure of explicit CFD codes. For this reason, the MPI version of P-SPH was structured in the same manner as the BSP code. The pseudo code with BSP is, therefore, the same as the MPI version shown in Figure 6 except that the group communication routines in BSP are non-blocking and so explicit barrier synchronisations are added. This structure is the basis of the BSP method, with a region of general computation followed by group communication and a barrier synchronisation forming a super step. The BSP version follows the simple parallel model, as with MPI, therefore, the whole particle array must be updated on each process.

5 Performance results

In order to investigate the performance of the P-SPH code in parallel, simulations of around one hundred time steps in length were run for sizes between 6,250 particles and 200,000 particles. Simulations were run sequentially and with one, two, four and eight processors in parallel using OpenMP, MPI and BSP. Two computers were used in the tests, a shared memory computer: an SGI Origin 2000 with 64 MIPS R10000 Processors, and a distributed memory computer: a Beowulf class workstation cluster with 33 dual processor Pentium III 800 MHz. In the tests OpenMP was used on the Origin 2000, with the SGI OpenMP libraries, MPI was used on both the Origin 2000, with the SGI Message Passing Toolkit (MPT) and the Beowulf cluster, with LAM-MPI, and BSP was used on the Origin 2000, with the Oxford University developed BSP library.

TABLE 1: Run times of P-SPH using OpenMP on the Origin 2000 (in seconds)

		Number of processors used by P-SPH			
N	Sequential	1	2	4	8
6250	6.58	6.89	4.62	3.04	2.44
12500	23.26	23.83	14.01	9.73	6.44
25000	78.37	79.10	45.20	27.30	18.22
50000	346.68	337.59	198.91	107.02	60.34
100000	1352.76	1301.76	717.43	384.93	232.81
200000	4805.65	4963.13	2672.26	1453.14	851.47

Table 1 shows the run time for the P-SPH code using OpenMP on the Origin 2000 as well as the sequential performance on an R10000 processor. This table shows how the run time is reduced by running the code in parallel as well as the scaling of the code run time with the number of particles. There is a certain amount of variation in the timing data due to the use of a shared system and network contention; however, this should not affect the trends used to analyse the scaling of the code.

The parallel efficiency is the ratio of the parallel run time to the sequential run time divided by the number of processors. Parallel efficiency shows the degree to which the incurred overheads, relative to the amount of parallel computation, have affected the performance. A code run in parallel on one processor will have an efficiency less than one since unnecessary work will be performed by the parallel library.

The OpenMP based code performed well for the range of simulation sizes. The efficiency, although low for the smallest simulations, increased as the number of particles in the simulation was increased.

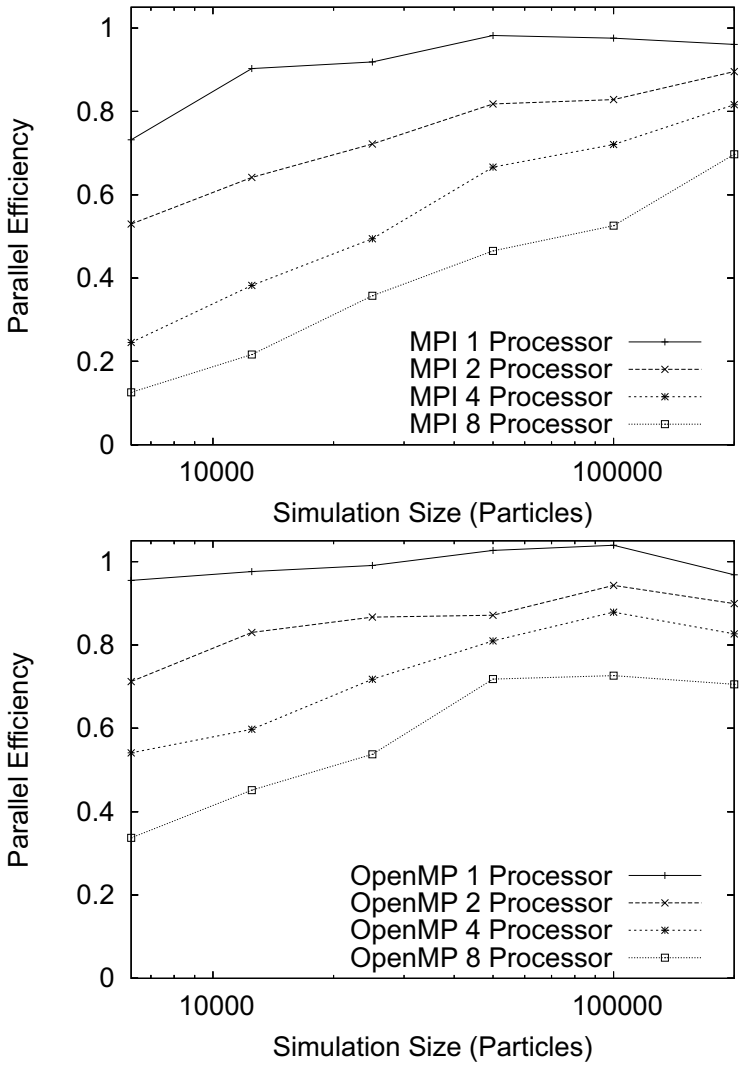

FIGURE 7: Effect of problem size N on parallel efficiency of P-SPH using MPI and OpenMP on the Origin 2000

On four processors, the speed-up was still well below the ideal of 4.0. However, for the code with little parallel optimization, and small N, a speed-up of 3.11 is a satisfactory outcome. The OpenMP code is useful as a baseline for what could be achieved with little coding effort using the shared memory model. The efficiency of OpenMP on the Origin 2000 for the range of simulation sizes and processor numbers is shown in Figure 7.

Message passing is inefficient with the small grained parallelism that results with small numbers of particles in simulations. Sending small messages through the interconnection network increases the effect of the system latency; latency is independent of message size and, therefore, has a lesser effect on the total send time for larger messages. For the larger simulations the run times scale along side the sequential run times. The parallel efficiencies for the MPI simulations are also shown in Figure 7. The efficiency improves greatly as the number of particles is increased as would be expected with the use of message passing.

The parallel performance of the code using OpenMP, MPI and BSP on the Origin 2000 can be compared directly for the ranges of simulation size using four processors. MPI and BSP, both being based on message passing, perform roughly the same over the range of simulation sizes. The performance of OpenMP is good for the whole range of sizes. The Origin 2000 was specially designed to take advantage of OpenMP code and this level of performance would be expected. The message passing libraries perform badly for the smaller simulation sizes mainly due to communication latencies associated with communication with small message sizes; OpenMP does not suffer from this problem. As the simulation size is increased the performance of the message passing libraries improves dramatically until they are equal with that of OpenMP.

The same general trends in scaling of efficiency are evident in

the MPI and the OpenMP based codes, however, message passing is inefficient for small simulations and its efficiency increases at a rapid rate and reaches that of OpenMP for the largest simulation size.

Varying the number of processors at the largest simulation size, of 200,000 particles, shows the decrease in parallel efficiency with increased number of processors, see Figure 8. MPI, with its message passing approach is well suited to the coarse grained parallelism of this large simulation size and performs the well through the range of processor numbers. OpenMP improves relative to message passing with an increased number of processors. As discussed earlier, OpenMP is better at the small simulation sizes.

The performance of MPI on the Beowulf cluster could not be examined properly due to faults in the Redhat Linux 7.0 operating system libraries on the system used and the use of a system shared with other users. Increasing the number of processors above two resulted in utilisation of the processors around 10% and so the performance results did not reflect the true parallel performance of the code on this system. For up to two processors, as seen in Figure 9, the same general trend is evident as that on the Origin 2000, as well as the relative speeds of the systems.

6 Conclusions

Where shared memory is available OpenMP will give good performance over the range of simulation sizes with moderate development. Message passing, using MPI and BSP, may require significantly more development to obtain good performance and is not suited to small simulation sizes. Using message passing does, however, allow distributed memory computers to be used. These computers scale more effectively and are usually cheaper than shared

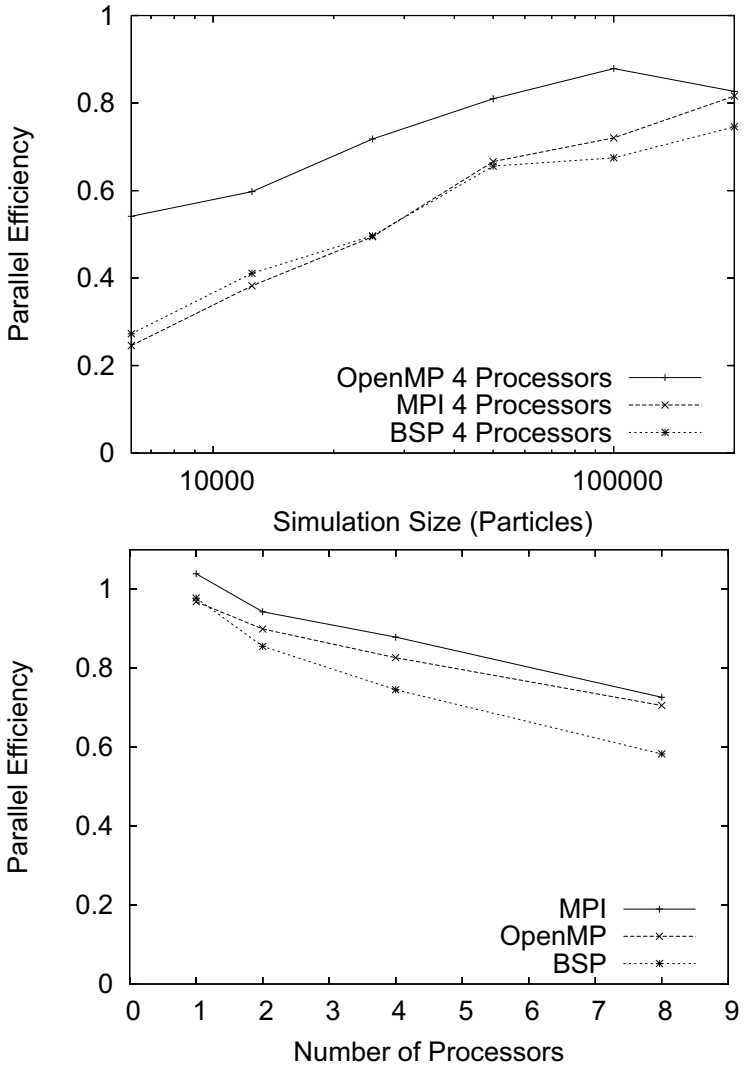

FIGURE 8: Parallel efficiency of MPI, OpenMP and BSP on the Origin 2000 for varying problem size (left) and processor numbers (right)

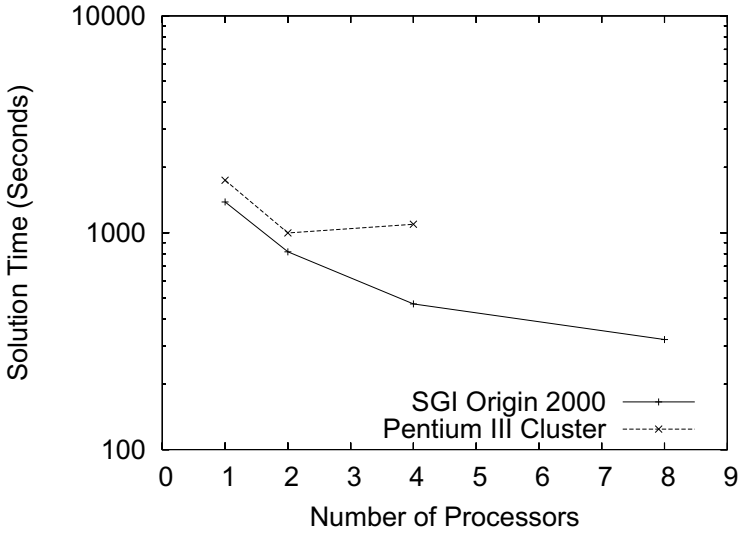

FIGURE 9: Comparison of the Performance of MPI on the Origin 2000 and a Beowulf Cluster

memory computers and, as a result, make up the majority of the world's 500 most powerful computers.

As expected, the efficiency increased with the simulation size, for a given number of processors, and the efficiency decreased with the number of processors, for a given simulation size. Parallel overheads were primarily due to communication and synchronization. The communication overhead can be broken down into the latency due to initiating the transfer and the actual communication time for the data. For smaller simulations, the system latency has a more significant effect on the communication time for message passing, as with MPI and BSP. Increasing the number of processors, although allowing access to more computing power, increases the overheads due to synchronising the processors, reducing the parallel efficiency.

References

[1] Touati Sid Ahmed Ali. MPI: A message passing interface. *MPI Forum*, pages 1–5, 1993. C217

[2] David A. Fulk and Dennis W. Quinn. An analysis of 1-D smoothed particle hydrodynamics kernels. *Journal of Computational Physics*, 126:165–180, 1996. C209

[3] R.A. Gingold and J.J Monaghan. Smoothed particle hydrodynamics: Theory and application to non-spherical stars. *Mon. Not. R. Astr. Soc.*, 181:375–389, 1977. C205

[4] William Gropp and Ewing Lusk. Installation guide to MPICH, a portable implementation of MPI. version 1.2.0. *ANL/MCS-TM-ANL-96/5 Rev B*, pages 1–63, 1996. C217

[5] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, Cambridge, Massachusetts, 1994. C217

[6] P. A. Jacobs. Numerical simulation of transient hypervelocity flow in an expansion tube. *Computers Fluids*, 23(1):77–101, 1994. C205

[7] L.B. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 82(12):1013–1024, 1977. C205

[8] J. J. Monaghan. An introduction to SPH. *Computers in Physics Communications*, 48:89–96, 1988. C208

[9] Inc. MPI Software Technology. *MPI/Pro Webpage*. http://www.mpi-softtech.com/, Accessed December 2000. C217

[10] University of Mannheim and University of Tennessee. *Top 500 Supercomputer Sites*. http://www.top500.org/, 2000. C213

[11] University of Notre Dame. *LAM/MPI Parallel Computing*. University of Notre Dame, 2000. http://www.lam-mpi.org/. C217

[12] L. G. Valiant. A bridging model for parallel computation. *Comm. of ACM*, 33(8):103–111, 1990. C219

[13] M. P. Wand and M. C. Jones. *Kernel Smoothing*. Chapman and Hall, London, 1995. C207